(19) Europäisches Patentamt
European Patent Office
Office européen des brevets

(11) EP 0 673 523 B1

(12) EUROPEAN PATENT SPECIFICATION

(45) Date of publication and mention of the grant of the patent:
10.02.1999 Bulletin 1999/06

(21) Application number: 93922031.5

(22) Date of filing: 08.10.1993

(51) Int Cl.6: G06F 9/46, G06F 11/14

(86) International application number:
PCT/GB93/02086

(87) International publication number:
WO 95/10805 (20.04.1995 Gazette 1995/17)

(54) MESSAGE TRANSMISSION ACROSS A NETWORK

NACHRICHTENÜBERTRAGUNG ÜBER EIN NETZWERK

TRANSMISSION DE MESSAGES SUR UN RESEAU

(84) Designated Contracting States:
AT BE CH DE ES FR GB IT LI NL SE

(43) Date of publication of application:
27.09.1995 Bulletin 1995/39

(73) Proprietor: INTERNATIONAL BUSINESS MACHINES CORPORATION
Armonk, NY 10504 (US)

(72) Inventors:
- CLARKE, Paul 21 Heath Road North Southampton Hampshire SO3 6PP (GB)
- JOHNSON, Peter 4 Malmesbury Gardens Winchester Hampshire SO22 5LE (GB)
- KINGSTON, William 25 Hiltingbury Road Eastleigh Hampshire SO5 1SR (GB)
- DREW, Robin, Miles Cary, NC 27511 (US)
- BLACK, George 5 Keats Close Winchester Hampshire SO21 3HF (GB)
- MELI, Roger 38 St. Stephens Road Weeke Hampshire SO22 6DE (GB)

(74) Representative: Burt, Roger James, Dr.
IBM United Kingdom Limited
Intellectual Property Department
Hursley Park
Winchester Hampshire SO21 2JN (GB)

(56) References cited:
EP-A- 0 457 108

- ACM TRANSACTIONS ON COMPUTER SYSTEMS. vol. 3, no. 3 , August 1985 , NEW YORK US pages 204 - 266 ROBERT E. STROM ET AL. 'Optimistic recovery in distributed systems'
- IBM TECHNICAL DISCLOSURE BULLETIN. vol. 33, no. 10A , March 1991 , NEW YORK US pages 362 - 366 XP110087 'Method for avoiding and repairing damage to distributed transactions in a coordinated resource recovery system'

Description

Field of The invention

[0001] The pr s nt invention relates to the saf deliv- ery of messages between application programs in a transaction-oriented data processing network, such that no messages are lost and none are delivered more than once.

Background To The Invention

[0002] It is known for updates to computer system resources (such as databases or file resources) to be made as a coordinated set of changes to two or more resources, such that either all of the changes take effect or none of them does. In this way, resources are prevented from being made inconsistent from each other. If one of the set of update operations fails then the others must also not take effect. A sequence of associated operations which transforms a consistent state of a recoverable resource into another consistent state (without necessarily preserving consistency at all intermediate points) is known as a "unit of work". Transaction processing is the execution of discrete units of work that access and update shared data. Logical points of consistency at which resource changes are synchronised within transaction execution (e.g. at termination) are called commit points or syncpoints (see below). An application ends a unit of work by declaring a syncpoint, or by the application terminating. The characteristic of a transaction being accomplished as a whole or not at all is known as "atomicity".

[0003] Atomicity of a transaction is known to be achieved by resource updates made within the transaction being held in-doubt (uncommitted) until a syncpoint is declared at completion of the transaction. That is, the resource updates are only made permanent and visible to applications other than the one which performed the updates on successful completion. If the transaction fails to complete successfully, then all changes that have been made to resources during the partial execution are removed - the transaction is said to rollback (or synonymously to backout), the resources being restored to the consistent state which existed before the transaction began. Any party (e.g. an application or resource manager) with an interest in the unit of work can cause a rollback when a syncpoint is declared by indicating unreadiness to commit.

[0004] A common problem in the provision of failure-tolerant data transmission is how to determine what stage has been reached in the transfer of messages that were in-doubt (i.e. had not been committed) when a failure occurred, to ensure that no messages are lost and none are sent more than once. Not all transaction systems remember the state of in-doubt messages.

[0005] The commit procedure is known as a "single-phase" procedure if only a single resourc manager (the system software which controls resources) is responsible for coordinating the commitment f changes made by the transaction. Singl phase commit processing is fficient in normal forward processing, consisting simply of issuance of a COMMIT peration instruction by an application or resource manager and then execution of the operation by the recipients of the instruction. There may be more than one resource manager involved, but the coordinator only calls each one once at syncpoint time to instruct them either to commit or rollback. In the vast majority of cases, all resource updates will be committed without error or interruption. However, if a problem arises (e.g. system or communication link failure) such that not all resource managers are unable to commit, then the resources can end up in an inconsistent state with some commits having been completed while others have not. The inconsistent resources then respire resynchronisation. The cost of rebuilding non-critical resources following such a problem may be tolerable in view of the efficiency of the single-phase commit procedure.

[0006] In contrast, a two-phase commit procedure is often required to protect critical resources from such inconsistencies. For example, a financial application to carry out a funds transfer from one account to another account has two basic operations no perform to critical resources: the debit of one account and the credit of the other. It is important to ensure that either both or neither of these operations take effect. A two-phase commit procedure under the control of a syncpoint manager consists of the following steps:

1. During a prepare phase, each participant resource is polled by the syncpoint manager to determine whether the resource is ready to confirm and finalise all changes. Each resource promises to commit the resource update if all resources indicate readiness (i.e if they successfully complete the prepare phase);

2. During a commit phase, the syncpoint manager instructs all resources to finalise the updates or to back them out if any resource could not complete the prepare phase successfully.

[0007] The advantage of the additional prepare phase is in reducing the likelihood of inconsistencies, but there remains a period during processing at which even two-phase commit leaves the possibility of inconsistencies between resources if an error occurs. Also, there is a cost which accompanies the two-phase commit's reduction in the probability of inconsistencies: since all updated resources must be locker to prevent further update access for the duration of the unit of work, additional steps in the commit processing may represent a considerable reduction in concurrency of resource update processing (partcularly if many resources are involved). If the resources are distributed around a network, two

phase commit requires a distributed unit of work, which introduces the likelihood of looks being held f r long p - riods, and also requires much mor c mplicated recovery procedur s. Three-phase and other multi-phase commit procedur s may be implemented t further reduce the window of time in which a failure can cause inconsistencies, but each additional step of preparation for commit represents a cost in loss of concurrency.

[0008] The IBM System Network Architecture SNA LU6.2 syncpoint architecture (reference SC31-6808 Chapter 5.2 "Presentation Services - Sync Point Verbs", published by International Business Machines Corporation) has been known to coordinate commits between two or more protected resources. This architecture addressed syncpoint facilities consisting of a syncpoint manager which performed both syncpoint and associated recovery processing running in a single application environment. Several applications could run simultaneously in this environment. The LU6.2 architecture supports a syncpoint manager (SPM) which is responsible for resource coordination, syncpoint logging and recovery.

[0009] According to the SNA LU6.2 architecture, in phase one and in phase two, commit procedures are executed and the syncpoint manager logs the phase in the syncpoint log. Also, the syncpoint manager logs an identification of a logical unit of work which is currently being processed. Such logging assists the syncpoint manager in resource recovery or resynchronisation in the event that a problem arises during the two-phase commit procedure (e.g. a problem such as failure of a communication path or failure in a resource manager). If such a problem arises subsequent to entering the two-phase commit procedure, the log is read and resource recovery processing takes place to bring the resources involved in the commit to a consistent state. This two phase commit procedure requires locks to be held across different computers using distributed units of work.

Summary Of The Invention

[0010] In a first aspect, the present invention provides a method of inter-program communication in a transaction-oriented data processing network wherein a sender program is responsible for sending messages from a first node of the network and a receiver program is responsible for receiving messages at a second node of the network, messages to be transmitted between the two nodes being sent from the sender program within a first syncpoint-manager-controlled unit of work and being received by the receiver program within a second syncpoint-manager-controlled unit of work such that the sending and receiving operations are held in-doubt (uncommitted) until resolution of the first and second units of work, respectively, characterised in that the first and second units of work are logically linked so that commit processing at resolution of the units of work comprises ither:

in response to successful receipt of the m ssages by the receiver program, committing said second unit of w rk, transmitting to th sender program a positive confirmation of receipt, and in response to the positive confirmation committing the first unit of work; or

in response to unsuccessful receipt of the messages, rolling back the second unit of work, transmitting to the sender program a negative confirmation of receipt, and in response to said negative confirmation backing out the first unit of work.

[0011] The present invention reduces the problem of the known single-phase commit procedures of failures during commit processing causing inconsistencies between resources that then require resynchronisation, and also avoids the undesirable increased locking of resources that is a feature of the extra prepare stage in the known two-phase commit procedures.

[0012] Preferably, if the confirmation from the receiving program is lost, due to a system or communication link failure, then the first unit of work remains in doubt. Log records which were written for each get and put operation performed by the sending and receiving programs are then examined to determine which operations have been committed at the receiving end thereby to determine which operations should be committed and which should be backed out at the sending end.

[0013] The present invention may be implemented in a network of computers wherein application programs communicate using messaging and queuing and wherein a message queue manager program is located at each computer in the network, the transmission between the aforesaid sender and receiver programs being transmission between respective queue manager programs. The nodes of the network are either message queue managers or computer systems on which one or more queue managers are located. Message transmission between queue managers involves a first queue manager getting an application-program-originated message from a queue and sending the message, and a second queue manager receiving the message and putting it onto a second queue (either for processing by a local application program, or for transmission to another queue manager if neither the first nor the second queue manager was the destination queue manager). Messaging and queuing is described in the document "IBM Messaging and Queuing Series - technical reference " (SC33-0850-01, 1993), and below in relation to an embodiment of the present invention.

[0014] It is preferred that each message-sending or message-receiving unit of work may include a plurality of messages, and that each confirmation of receipt (or receipt and storage, if received messages are put to queues) may relate to such a plurality of messages. This method of transmitting messages in a batch as a unit of

work provides a great improvement in processing efficiency, since the transport connection direction (f r-wards f r m ssage transmission and backwards for confirmation of rec ipt) is only turned ar und at the end of each batch. This is distinguished from the prior art method of sending messages to queues as individual units of work and committing after each send operation, which risks leaving resources at the sending end in an inconsistent state with resources at the receiving end, and requires a change of direction of message flow after each send and after each confirmation if two phase commit is used. This batch transmission of messages between sending and receiving programs, as a stage of the transfer of messages between application programs, is also clearly distinguished from batch processing by an application program, which is well known in the art.

[0015] The messages which may be transmitted as a batch in this way may be logically unrelated and may be destined for different application programs (which may be served by different queue managers) - the only common factor between the messages which is necessary for them to be transmitted as a batch between a first and a second queue manager is that the second queue manager is the next queue manager from the first queue manager on the way to each message's destination queue manager. Prior art methods of message transmission do not enable batch transmission (where batch size is greater then one) of messages which are destined for different application programs, and so cannot benefit from the processing efficiency provided by the present invention. For many database systems, commit processing is the expensive stage of the processing in terms of computing facilities - in particular, disk access is expensive as compared with RAM processing - so improvements to commit processing efficiency are highly desirable.

[0016] Preferably, the batch has a request to commit the batch and to confirm receipt transmitted with it to the receiving program, so that commit processing is being coordinated by the sending end of the communication. A message may be transmitted as a plurality of segments if it is too large for the transport connection to transfer in one go. Where there is segmentation, the request for confirmation will be associated with the last segment in the batch. On successful receipt of the batch of messages at the receiving end the confirm request is acted on by committing the receipt and communicating a confirmation of the successful receipt.

[0017] In a second aspect, the present invention provides a method of inter-program communication in a transaction-oriented data processing network wherein a message to be delivered is sent to a queue from a sending application program at a first computer and is then asynchronously taken from the queue to be processed by a rec iving application program, characterised in that:

ach step f sending a message to a queue or taking a message from a queue is carried out under th control f a message queue manag r program, at least one of which is located at each computer in th network;

messages to be delivered to a local application program are put on a local queue serviced by the local application program; whereas

messages to be delivered to remote application programs on remote computers are put on local transmission queues for transmission, respectively for each transmission queue, to the next message queue manager program on the way to the respective destination remote message queue manager programs, wherein all messages put on a particular transmission queue, which messages may be destined for different destination message queue manager programs, are transmissable to said next message queue manager as a batch of messages within a syncpoint-manager-controlled unit of work.

<u>Description of an Embodiment</u>

[0018] The present invention will now be described in more detail with reference to the accompanying drawings in which:

Figure 1 is a representation of the data fields making up a message;

Figure 2 is a schematic representation of two programs communicating with each other using messaging and queuing;

Figure 3 is a representation of two adjacent computer systems and the interrelationships between the system entities involved in message communication according to an embodiment of the present invention;

Figure 4 is an overview flow diagram of a method of message communication between application programs according to an embodiment of the present invention;

Figure 5 is a representation of the message flows between processes during normal forward processing in a method of communication between application programs, according to an embodiment of the present invention.

[0019] Message queuing is a message of inter-program communication which allows programs to send and receive application-specific data without having a direct connection established between them. Before describing the detail of a specific implementation of the present invention in a messaging and queuing network, it will be helpful to describe the general methodology of inter-program communication using messaging and queuing.

[0020] A message consists of two parts, application data 1 and a message descriptor 2 containing control

information 3, as shown in Figure 1. The application data in a message is defined and supplied by the application which sends the message. Ther are n constraints on the nature of the data in the message (for example, it could consist f one r more of bit strings, character strings, binary integers, packed decimal integers, floating point numbers). Applications view the string of bits and bytes that make up a message as consisting of a sequence of items which each have a particular data type and meaning. For example, if the message relates to a financial transaction, the first item 4 may be a four-byte unsigned binary integer containing an account number and the second item 5 may be a twenty-byte character string containing a customer name. This data is called the application data.

[0021] In addition to the application data, a message has associated with it some ancillary data. This is information that specifies the properties of the message, and is used by the message queuing service to decide how the message should be processed. Some of this information must be specified by the application. This ancillary control information is contained in a data structure called the message descriptor 2.

[0022] A message queue is a named object in which messages accumulate and from which they are later removed. Each queue belongs to one particular queue manager, which is responsible for the maintenance of that queue. A queue manager can own many queues, but each queue must have a name that is unique within the queue manager instance that owns the queue. A message queue is not merely a stack: when messages are added to a queue, they are added at the end, and when messages are taken from a queue, they are normally removed from the front (although facilities do exist for reading messages in other than a FIFO order - for example it may be desirable for messages which require a reply to be retrieved as a high priority).

[0023] The physical representation of a message queue depends on the environment, but can be a buffer or buffers in main storage, a file or files on disk or other permanent storage device, or both of these. However, the physical management of message queues is entirely the responsibility of a queue manager (the system service that provides the message-queuing facilities used by applications), and such details are not made apparent to the application program. Applications can view a message queue simply as a "black box" in which messages accumulate. Applications have no access to message queues other than through the message queuing calls (such as MQGET for taking messages from a queue and MQPUT for sending messages to a queue). Applications obtain message queuing services by using the message-queuing calls to communicate with the queue manager that is installed on the same system as the application (i.e the local queue manager).

[0024] For message queuing services to be available, there must be at least one queue manager on a system. More than one queue manager may b required, for example, in order to k p development work separat from production work. Each different queue manager instance is known by its name, which must g nerally b unique within the network of interconn cted queue managers so that one queu manager can unambiguously identify the target queue manager to which any given message should be sent.

[0025] Applications communicate by agreeing to use particular named message queues, sending messages to the specific target queues that the application programs have agreed to read from. The location of these queues need not be apparent to the applications which send the messages; each application interacts only with its local queue manager, and it is the network of interconnected queue managers that is responsible for moving the messages to the intended queues. Since cross-network communication sessions are established between queue managers rather than between individual programs, programs are less vulnerable to network failures than in certain other types of inter-program communication. If a link between processors fails, it is the job of the queue managers to recover from the failure. Programs on the effected processors are not brought to a halt by such an event, and indeed need not be aware that it has happened.

[0026] Figure 2 is a representation of the flow of messages between two communicating programs in a message queuing network in the simple example of one-to-one communication. The two programs 10,20 send messages to each other via queues 30,40 under the control of respective queue managers 50,60. The first program 10 puts messages onto the second program's queue 30 without a dedicated logical connection having to be established between the programs (this message flow is represented in Figure 2 by arrows f1, f2, f3 and f4). The queue managers 50,60 ensure that the messages are moved across the network, such that the programs themselves are shielded from network variations and complexities. This is represented in Figure 2 by network link 70. All of the work involved in maintaining message queues, in handling network failures and restarts, and in moving messages around the network, can be handled by the queue managers. Program 20 subsequently takes the messages from the queue 30 to process them, when it is ready rather than when the sending program 10 chooses. Any changes made to recoverable resources by the transfer cf messages and subsequent processing are recorded in recovery logs 80,90 for use in the event of a subsequent failure.

[0027] As represented in Figure 3, queue managers 100 may store messages onto a number of different queues. If the messages are eventually to be processed by local application programs then the queue manager stores them on local destination queues 110; and if the messages are eventually to be processed by a remote application, then the qu ue manager stores them in special local queues known as transmission queues 120. Transmission queues containing messages to be s nt

to queues belonging to remote queue managers enabl the movement of messages to r mote queues to be carried out in stages between adjacent queue managers. This staging of m ssage transmission, which will be described in detail below, is invisible to the application programs involved in the communication. There may be a plurality of local destination queues and of transmission queues controlled by a particular queue manager, as will be explained below.

[0028] The messages on a transmission queue are extended by the queue manager to include a transmission queue header in addition to the application message (the data being transferred by an application). The transmission queue header is an architected descriptor containing the name of the destination queue and the message descriptor. Messages on destination queues include the application data and a message header specifying control information.

[0029] The transport relationship between two queue managers is known as a channel. The key elements defining a channel are the name of a transmission queue, information concerning the transport processes or programs 130,150 which send or receive messages over the channel (these processes, which are part of the queue managers, are known as message channel agents - hereafter MCAs), and communications protocol and target system information for the destination to which messages on the transmission queue are to be sent. The association between a particular channel definition and the various data model entities involved in the message communication is represented by broken lines in Figure 3. Each named channel is defined in both the sending and receiving nodes. The channel name is used in the transmissions between the sender and receiver processes to identify the channel to the receiver or for a receiver to request that messages from a particular channel be sent. Channel definition has some information which is common for all environments and some which depends on the operating system environment and underlying communications protocol to be used.

[0030] The communication of messages between queue managers is carried out by MCAs working in pairs across specific channels: one sender 130 and one receiver 150. A pair of MCA processes uses a transport connection 170 such as a VTAM APPC session or a TCP/IP connection as a transport layer. Message traffic in the opposite direction flows between a sender 160 and a receiver 140 on a different channel, the channels being used effectively as uni-directional pipes between nodes. There are four types of MCAs:

Sender -	which takes messages from a transmission queue and sends them to a Receiver or Requester;
Receiver -	which receiv s messages and queues them;
Requester -	which s nds a single message to cause a Sender or S rver to be started remotely;
Server -	which is started by a message from a requester, and then b comes a Sender.

[0031] An MCA 130 dequeues messages from transmission queues and transmits them over the transport connection 170. The receiving MCA 150 queues the messages to the destination queues 180 named in the message header. These two units of work, dequeue and enqueue, are performed such that any failure at any point in the protocol can be detected and rectified so that each message is delivered once and once only. In the case where the destination queue is more than one hop from the original transmission queue, the receiving MCA will queue the message on another transmission queue for the next hop. This provides a safe store and, in the event that the next connection is unavailable, the necessary asynchronism to allow this first stage of transmission to still be carried out. The message format and the safe movement protocol are transport layer independent so that MCAs can support different transport protocols on different channels. The protocols used by the MCAs are described below.

[0032] A channel may be started in a number of different ways:

1. a terminal operator may issue a START CHANNEL command;
2. the channel can be triggered, a Sender MCA being started automatically by a queue manager when a message arrives on the transmission queue; or
3. by a network request - the communications transport being configured to automatically start an MCA when a request from the network is received. Receiver, Server and Sender channels could be configured this way.

[0033] Before any messages or data can flow down a channel, the two MCAs which are to use it must first negotiate the way in which they are going to communicate. Thus, channel initialisation involves negotiation of certain protocol parameters, such as which communication partner is going to do any needed conversion of control and message header data. Two MCAs may be running on systems using two different data formats. For example, one may be using ASCII and the other EBCDIC. One may be encoding numbers left to right, the other right to left. The control information and message header data must be converted from the sender's representation to the receiver's. Data conversion over channels applies only, to control information (such as destination queue name, control field Lengths, and the like): no application data conversion is performed by MCAs, since MCAs do not need to interact with the application data in a message when they transmit it.

[0034] The method of delivering messages between applications on different comput r syst ms involv s the

following steps, described with reference to Figures 4 and 5:

[0035] An application sends a message to a target destination queue for processing by another application by issuing (200) an MQPUT command. The local queue manager reads the destination queue name specified by the application in the message's header and determines (210) where to put the message. If the destination queue is a local queue then the local queue manager puts (220) the message into that local queue. The unit of work including the operation of putting the message to a queue must be committed before the message is available to other applications. An application serving that local queue can then asynchronously issue MQGET (230) to take the message from the queue for processing. The MQPUT and MQGET operations are within two separate units of work.

[0036] If the destination queue is not the responsibility of the local queue manager, then the local queue manager puts the message onto a local transmission queue (240), for transfer to another queue manager. There may be a plurality of transmission queues defined for each queue manager, but a one-to-one correspondence between transmission queues and remote destination queues is not necessary. All messages that are to be passed between two adjacent queue managers (that is, all messages to be sent from a first queue manager which have a common nearest neighbour queue manager in the direction of their respective target destination queue managers) can be put in the same transmission queue. It is equally possible to have a number of transmission queues for traffic going to the same next node. A maximum batch size is specified (for example 50 messages) to limit the number of messages which will have to be resent in the event of a failure. The unit of work 300 which puts the message to the transmission queue must be committed before the message is available to other processes.

[0037] The local queue manager (or an end user) starts a sender MCA to transmit messages to the next queue manager. The sender MCA then gets messages (250) (issues MQGET) from a transmission queue owned by this queue manager and transmits them as a batch to the next queue manager on the way to the destination queue manager or queue managers. Each message is either transmitted in one transmission or as a plurality of transmission segments in a plurality of transmissions if the messages are too large for the transport connection to send in one go (e.g. a message might be 4 Megabytes in size and the maximum transfer size 32 kilobytes). The steps of getting and transmitting messages is performed within a syncpoint-manager-controlled unit of work 330, which is held in-doubt by the sender at this stage. Log records are written specifying the in-doubt state of the resource updates. The batch has a request for confirmation of rec ipt of the batch attached to it: this is implemented by the last message (or the last transmission segment of the last message) of the batch having a Request_Confirm control flag set in its transmission segment header.

[0038] Each message has a m ssage sequenc number associated with it - on of a monotonically increasing sequence of numbers, uniquely assigned t a single application message on a channel. Message sequence numbers are used to resynchronise between sender and receiver in the event of a link failure or program failure. The highest message sequence number in the batch is taken as the logical unit of work identifier (LUWID) - a unique value defining a batch of messages on a channel which are under control of a syncpoint manager.

[0039] The receiver MCA receives (260) the messages and the receiver queue manager determines (210) where each message is to be sent (as the sending queue manager program did previously). The receiver queue manager puts the messages (using MQPUT) within a syncpoint-manager-controlled unit of work 360 to queues belonging to the receiving computer system's queue manager, which may be the actual application-specified destination queue for a particular message or may be a related transmission queue for the next hop towards the target system.

[0040] Either all of the messages in the batch of messages transferred by MCAs are successfully received and queued by the receiving queue manager or the batch is rejected as a whole and not safe stored at the receiver (the unit of work is rolled back). If the batch is successfully received and queued then the receiver sends an acknowledgement of receipt and storage (a Status segment indicating "No error" is transmitted), having logged the LUWID and committed the batch of messages together as an atomic action. On receipt of the positive acknowledgement the sender also commits the batch of messages using the LUWID, this commit of the MQGET operation deleting the messages from the transmission queue. The next batch can then be started. If no messages are left on the transmission queue (and a preset time interval has expired) or a request to close the channel has been received, then the connection can be terminated.

[0041] If the batch is rejected, an acknowledgement of rejection (a Status segment indicating Error - which may include details of the error) is transmitted to the sender which then rolls back its in-doubt messages onto the transmission queue ready for retry, and terminates the channel. If a batch of messages is rolled back, the sequence number or LUWID must also be rolled back, to the value of the last successfully committed batch. If no confirmation is received, due to transport or communication-partner failure, then the channel is terminated by the sender and the receiver MCA's unit of work is rolled back. If the sender has not yet sent a confirm request then the sender MCA should also roll back. If it has sent a confirm request then its log records and those of the receiver program must be examined to determine whether it should be committed or rolled back. The

MCAs automatically perform the determination of whether the first unit of work should be committed or rolled back (unless contact cannot be reestablished in which cas the operator may take the decision). Following a r llback, th sending MCA may try t re-establish a channel and resynchronise with the sending MCA in order to resend the failed batch.

[0042] Channel resynchronisation is achieved during channel initialisation. The sender MCA retrieves from its log the in-doubt LUWID, or message sequence number of the last message sent for which a confirmation was also sent. The receiving MCA will check his logged LUWIDs or sequence numbers to determine whether he is in sync with the sender. As a result of the comparison, he will confirm or reject the resynchronisation request by returning an appropriate Status segment, containing the LUWID or sequence number of the last successfully committed message or batch of messages at his end. If this value matches the sender's, the sender may commit the previously sent messages, and commence sending the next one. If the receiver's value matches the previous LUWID or sequence number, the sender rolls back and resends the previous message or batch.

[0043] The MCAs thus use a syncpoint manager to control each batch as a logical unit of work. The unit of work including the MQGET of the sender message queue manager and the unit of work including the MQPUT of the receiver message queue manager are logically linked in that both are held in doubt until the receiver is ready to commit, messages being committed at the receiving end before deleting them at the sending end using a single-phase commit protocol. Two phase commit is not required as the sender acts as a commit coordinator. Any system failure that occurs before the end of the batch, either at the sender or receiver, may require the unit of work to be backed out during a resynchronisation phase.

[0044] This single-phase commit using logical linkage of units cf work on different systems avoids the problem of a two phase commit needing to synchronise (lock) all participating resources in a distributed unit of work. In the present invention, resource managers do not actually have to synchronise with each other. A limited period of inconsistency between resources as viewed by applications is accepted, but final consistency is assured since atomic transaction processing is assured.

[0045] To complete the assured delivery of messages, the target application which services the destination queue can issue MQGET to get messages from the queue as part of a unit of work 390 under the control of its local syncpoint manager, to allow rollback of the message to the queue in case of application failure or commit of a successfully processed message to delete it.

Claims

1. A method of inter-program communication in a transaction-oriented data processing network wherein a sender program is responsible for sending messages from a first node f the n twork and a r ceiver program is responsible for receiving messages at a second node of the network, messages to be transmitted between the two nodes being sent by the sender program within a first syncpoint-manager-controlled unit of work and being received by the receiver program within a second syncpoint-manager-controlled unit of work such that the sending and receiving operations are held in-doubt, i.e. uncommitted, until resolution of the first and second units of work, respectively, characterised in that the first and second units of work are logically linked so that commit processing at resolution of the units of work comprises either:

   in response to successful receipt of the messages by the receiver program, committing said second unit of work, transmitting to the sender program a positive confirmation of receipt, and in response to the positive confirmation committing the first unit of work; or

   in response to unsuccessful receipt of the messages, rolling back the second unit of work, transmitting to the sender program a negative confirmation of receipt, and in response to said negative confirmation backing out the first unit of work.

2. A method according to claim 1 wherein said sender and receiver programs are located on adjacent nodes within a network, and wherein messages, which may be destined for different destination nodes, are transmitted between adjacent nodes on the way to their respective destination nodes as a batch of messages within a unit of work, the units of work incorporating said sending and receiving operations being held in-doubt until the end of the batch.

3. A method according to claim 2 wherein the last message in a batch is transmitted together with a request for commitment of and for confirmation of receipt of the batch, the commitment of said second unit of work and the transmission of said positive or negative confirmation being in response to said request.

4. A method according to any one of claims 1 to 3 wherein log records are written to record the in-doubt status of said units of work for use in recovery processing following a failure during the processing of said units of work, the log records being read during recovery processing to determine which units of work should be committed and which should be backed out.

5. A method of inter-program communication according to any one of the preceding claims, which implements messaging and queuing for communication b tween application programs, the application programs sending messages to messag queues from where receiver application programs can asynchronously take the messages for processing or forwarding on.

6. A method according to claim 5, wherein communication between application programs running on different computer systems of the network comprises at least the following steps:

a first application program issuing a put message instruction under control of a synchpoint manager in the sending computer system, for sending a message to a message queue;

sender and receiver transmission programs transferring messages between the computer systems, as two logically linked units of work, using synchpoint managers in both the sending and receiving computer systems; and

a second application program issuing a get message instruction under control of a synchpoint manager in the receiving computer system, for taking the message from the queue;

wherein the units of work of put message, transfer and get message are each held in-doubt until resolution of the respective unit of work.

7. A method according to claim 1, wherein a message to be delivered is sent to a queue from a sending application program at a first computer and is then asynchronously taken from the queue to be processed by a receiving application program, characterised in that:

each step of sending a message to a Queue or taking a message from a queue is carried out under the control of a message queue manager program, located at a computer in the network;

messages to be delivered to a local application program are put on a local queue serviced by the local application program; whereas

messages to be delivered to remote application programs on remote computers are put on local transmission queues for transmission, respectively for each transmission queue, to the next message queue manager program on the way to the resp ctive destination remote message queue manager programs, wherein all m ssages put on a particular transmission queue, which messages may be destined for different destination message queue manager programs, are transmissible to said next m ssag queue manager as one or more batches of messages within syncp int-manager-controlled units of work.

8. A data processing system including a messaging manager for inter-program communication across a network of data processing systems, the messaging manager including sender and receiver programs for transferring messages between adjacent messaging managers in the network in accordance with the following transfer protocol:

a sender program of a first messaging manager sending one or more messages within a first syncpoint-manager-controlled unit of work;

a receiver program in a second messaging manager receiving said messages within a second syncpoint-manager-controlled unit of work;

the sending and receiving operations being held in-doub, i.e. uncommitted, until resolution of the first and second units of work, respectively; and

characterized in that the first and second units of work being logically linked so that commit processing at resolution of the first and second units of work comprises either

(i) in response to successful receipt of the messages by the receiver program, committing said second unit of work, transmitting to the sender program a positive confirmation of receipt, and in response to the positive confirmation committing the first unit of work; or
(ii) in response to unsuccessful receipt of the messages, rolling back the second unit of work, transmitting to the sender program a negative confirmation of receipt, and in response to said negative confirmation backing out the first unit of work.

9. A data processing system according to claim 8, wherein the messaging manager is adapted for message queuing inter-program communication across a heterogeneous network of data processing systems, the messaging manager including an application programming interface by which applications attach to the messaging manager and providing queuing services enabling application programs to put messages onto message queues for asynchronous retrieval by other application programs.

Patentansprüche

1. Ein Verfahren der Interprogramm-Kommunikation in einem transaktionsorientierten Datenverarbeitungsnetzwerk, w bei ein Senderprogramm verantwortlich ist, Nachrichten von einem ersten Knoten des Netzwerks zu senden, und ein Empfängerprogramm verantwortlich ist, Nachrichten in einem zweiten Knoten des Netzwerks zu empfangen, Nachrichten zwischen den beiden Knoten zu übertragen, die von dem Senderprogramm innerhalb einer ersten, vom Synchronisierungspunkt-Manager gesteuerten Arbeitseinheit gesendet und von dem Empfängerprogramm innerhalb einer zweiten, vom Synchronisierungspunkt-Manager gesteuerten Arbeitseinheit empfangen wurden, so daß die Sende- und Empfangsoperationen bis zur Auflösung der ersten bzw. der zweiten Arbeitseinheit in Zweifel gezogen werden, d.h. nicht festgeschrieben werden, dadurch gekennzeichnet, daß die ersten und zweiten Arbeitseinheiten logisch verknüpft sind, so daß die Festschreibverarbeitung in der Auflösung der Arbeitseinheiten entweder enthält:

   als Reaktion auf den erfolgreichen Empfang der Nachrichten von dem Empfängerprogramm, die zweite Arbeitseinheit festzuschreiben, an das Senderprogramm eine positive Empfangsbestätigung zu senden, und als Reaktion auf die positive Bestätigung die erste Arbeitseinheit festzuschreiben; oder um

   als Reaktion auf den nicht erfolgreichen Empfang der Nachrichten, die zweite Arbeitseinheit zu wiederholen, dem Senderprogramm eine negative Empfangsbestätigung zu senden, und als Reaktion auf die negative Bestätigung, die erste Arbeitseinheit zurückzusetzen.

2. Ein Verfahren gemäß Anspruch 1, wobei die Sender- und Empfängerprogramme in benachbarten Knoten in einem Netzwerk untergebracht sind, und wobei Nachrichten, die für verschiedene Zielknoten bestimmt sein können, zwischen den benachbarten Knoten über ihre jeweiligen Zielknoten als ein Stapel von Nachrichten innerhalb einer Arbeitseinheit übertragen werden, wobei die Arbeitseinheiten, welche die Sende- und Empfangsoperationen enthalten, bis zum Ende des Stapels in Zweifel gezogen werden.

3. Ein Verfahren gemäß Anspruch 2, wobei die letzte Nachricht in einem Stapel zusammen mit einer Anforderung übertragen wird, den Empfang des Stapels festzuschreiben und zu bestätigen, wobei das Festschreiben der zweiten Arbeits inheit und die Übertragung der positiven und negativen Bestätigung eine Reaktion auf di se Anforderung ist.

4. Ein Verfahren gemäß einem der Ansprüche 1 bis 3, wobei die Protokollaufzeichnungen geschrieben werden, um den zw ifelhaft n Status der Arbeitseinheiten aufzuzeichnen, um diese für die Wied rherstellungsv rarbeitung im Anschluß an einen Ausfall während der Verarbeitung der Arbeitseinheiten zu benutzen, wobei die Protokollaufzeichnungen während der Wiederherstellungsverarbeitung gelesen werden, um zu bestimmen, welche Arbeitseinheiten festgeschrieben und welche zurückgesetzt werden sollten.

5. Ein Verfahren der Interprogramm-Kommunikation gemäß einem der vorhergehenden Ansprüche, welche die Nachrichtenerstellung und die Warteschlangenbildung zur Kommunikation zwischen den Anwendungsprogrammen implementieren, wobei die Anwendungsprogramme Nachrichten an Nachrichten-Warteschlangen senden, aus denen die Empfängeranwendungsprogramme Nachrichten asynchron herausnehmen können, um diese zu verarbeiten oder zu senden.

6. Ein Verfahren gemäß Anspruch 5, wobei die Kommunikation zwischen den Anwendungsprogrammen, die in verschiedenen Computersystemen des Netzwerks laufen, wenigstens die folgenden Schritte enthält:

   ein erstes Anwendungsprogramm, das einen Put-Nachrichtenbefehl unter der Kontrolle eines SynchronisierungspunktManagers in das sendende Computersystem eingibt, um eine Nachricht an eine Nachrichten-Warteschlange zu senden;

   Sender- und Empfängerübertragungsprogramme, die Nachrichten zwischen den Computersystemen wie zwei logisch verknüpfte Arbeitseinheiten mittels Synchronisierungspunkt-Managern sowohl an die sendenden als auch die empfangenden Computersysteme übertragen; und

   ein zweites Anwendungsprogramm, das einen Get-Nachrichtenbefehl unter der Steuerung eines Synchronisierungspunkt-Managers an das empfangende Computersystem ausgibt, um die Nachricht aus der Warteschlange zu nehmen;

   wobei die Arbeitseinheiten der Put-Nachricht, der Transfer- und Get-Nachricht jeweils bis zur Auflösung der jeweiligen Arbeitseinheit angezweifelt werden.

7. Ein Verfahren gemäß Anspruch 1, wob i eine auszuhändigende Nachricht von einem sendenden An-

wendungsprogramm in einem ersten Computer an eine Warteschlange g send t wird und dann asynchron von einem Empfangsanwendungsprogramm aus der zu v rarb itenden Warteschlange herausgenommen wird, wobei das Verfahren dadurch g - kennzeichnet ist, daß

jeder Schritt, um eine Nachricht an eine Warteschlange zu senden oder um eine Nachricht aus einer Warteschlange herauszunehmen, unter der Kontrolle eines Nachrichten-Warteschlangen-Managerprogramms erfolgt, das in einem Computer im Netzwerk vorhanden ist;

Nachrichten, die an ein lokales Anwendungsprogramm zu liefern sind, in eine lokale Warteschlange gelegt werden, die von dem lokalen Anwendungsprogramm bedient wird; während

Nachrichten, die an Fernanwendungsprogramme in entfernt stehenden Computern zu liefern sind, in lokale Übertragungswarteschlangen für die Übertragung gelegt werden bzw. für jede Übertragungswarteschlange an das nächste Nachrichten-Warteschlangen-Managerprogramm über das jeweilige entfernt stehende Ziel-Nachrichten-Warteschlangen-Managerprogramm, wobei alle Nachrichten, die in eine bestimmte Übertragungswarteschlange gelegt werden, deren Nachrichten für verschiedene Ziel-Nachrichten-Warteschlangen-Managerprogramme bestimmt sind, an das nächste Nachrichten-Warteschlangen-Managerprogramm als ein Stapel Nachrichten in einer von einem Synchronisierungspunkt-Manager gesteuerten Arbeitseinheit übertragbar sind.

8. Ein Datenverarbeitungssystem, das einen Nachrichtenerstellungsmanager zur Interprogramm-Kommunikation innerhalb eines Netzwerks mit Datenverarbeitungssystemen hat, wobei der Nachrichtenerstellungsmanager Sender- und Empfängerprogramme enthält, um Nachrichten zwischen benachbarten Nachrichtenerstellungsmanagern innerhalb des Netzwerks gemäß dem folgenden Übertragungsprotokoll zu übertragen:

ein Senderprogramm mit einem ersten Nachrichtenerstellungsmanager, der eine oder mehrere Nachrichten innerhalb einer ersten, von einem Synchronisierungspunkt-Manager gesteuerten Arbeitseinheit sendet;

ein Empfängerprogramm in einem zweiten Nachrichtenerstellungsmanager, der Nachrichten innerhalb iner zweiten, v n einem Synchronisierungspunkt-Manager gesteuerten Arbeitseinheit empfängt;

wobei die Sende- und Empfangsoperationen bis zur Auflösung der ersten bzw. der zweiten Arbeitseinheit angezweif lt werden, d.h. nicht festgeschrieben werden; und

dadurch gekennzeichnet, daß die ersten und zweiten Arbeitseinheiten logisch verknüpft sind, so daß die Festschreibverarbeitung in der Auflösung der Arbeitseinheiten entweder enthält:

(i) als Reaktion auf den erfolgreichen Empfang der Nachrichten von dem Empfängerprogramm, die zweite Arbeitseinheit festzuschreiben, an das Senderprogramm eine positive Empfangsbestätigung zu senden, und als Reaktion auf die positive Bestätigung die erste Arbeitseinheit festzuschreiben; oder um

(ii) als Reaktion auf den nicht erfolgreichen Empfang der Nachrichten, die zweite Arbeitseinheit zu wiederholen, dem Senderprogramm eine negative Empfangsbestätigung zu senden, und als Reaktion auf die negative Bestätigung, die erste Arbeitseinheit zurückzusetzen.

9. Ein Datenverarbeitungssystem gemäß Anspruch 8, wobei der Nachrichtenerstellungsmanager angepaßt ist, um Nachrichten in der Interprogramm-Kommunikation innerhalb eines heterogenen Netzwerks von Datenverarbeitungssystemen in Warteschlangen einzureihen, und der Nachrichtenerstellungsmanager eine Anwendungsprogrammschnittstelle enthält, über die Anwendungen mit dem Nachrichtenerstellungsmanager verbunden werden und Warteschlangenbildungs-Services bereitgestellt werden, mit denen Anwendungsprogramme Nachrichten in Nachrichten-Warteschlangen zwecks asynchronem Abruf durch andere Anwendungsprogramme legen können.

Revendications

1. Procédé de communications entre des programmes dans un réseau de traitement de données orienté transactions dans lequel un programme émetteur est responsable de l'émission de messages à partir d'un premier noeud du réseau et un programme récepteur est responsable de la réception des messages au niveau d'un second noeud du réseau, les messages devant être transmis entre les deux noeuds étant émis par le programme émetteur à l'intérieur d'une première unité de tâche commandée par un gestionnaire de point de synchronisation et étant reçus par le programme récepteur à l'intérieur d'une seconde unité de tâche commandée par un gestionnaire de point de synchronisation de sor-

te que les opérations d'émission et de réception soient maintenues dans l'incertitude, c'est-à-dire non ngagées, jusqu'à la résolution des première et seconde unités de tâche, r spectivement, caractérisé en ce que les pr mièr et seconde unités de tâche sont liées logiquement de sorte que l'engagement du traitement à la résolution des unités de tâche comprend soit :

en réponse à la réception correcte des messages par le programme récepteur, l'engagement de ladite seconde unité de tâche, la transmission vers le programme émetteur d'une confirmation de réception positive, et en réponse à la confirmation positive l'engagement de la première unité de tâche, ou

en réponse à une réception incorrecte des messages, le renvoi de la seconde unité de tâche, en transmettant au programme émetteur une confirmation de réception négative, et en réponse à la confirmation négative la restitution de la première unité de tâche.

2. Procédé selon la revendication 1, dans lequel lesdits programmes émetteur et récepteur sont localisés sur des noeuds adjacents à l'intérieur d'un réseau, et dans lequel des messages, qui peuvent être destinés à différents noeuds de destination, sont transmis entre des noeuds adjacents en route vers leurs noeuds de destination respectifs sous forme d'un lot de messages à l'intérieur d'une unité de tâche, les unités de tâche incorporant lesdites opérations d'émission et de réception qui sont maintenues dans l'incertitude jusqu'à la fin du lot.

3. Procédé selon la revendication 2, dans lequel le dernier message d'un lot est transmis en même temps qu'une demande d'engagement et pour confirmation de la réception du lot, l'engagement de ladite seconde unité de tâche et la transmission de ladite confirmation positive ou négative se faisant en réponse à ladite demande.

4. Procédé selon l'une quelconque des revendications 1 à 3, dans lequel des enregistrements de journalisation sont écrits pour enregistrer la situation d'incertitude desdites unités de tâche en vue d'une utilisation dans le traitement de récupération qui suit une défaillance durant le traitement desdites unités de tâche, les enregistrements de journalisation étant lus pendant le traitement de récupération afin de déterminer quelles unités de tâche devraient être engagées et lesquelles devraient être restituées.

5. Procédé de communications entre des programmes selon l'une qu lconque des revendications précédentes, qui met en oeuvre une messagerie t une mise en file d'attente des communications entre des programm s d'application, les programmes d'application émettant d s messages vers des fil s d'attente de m ssages à partir desquelles des programmes d'application du récepteur peuvent prélever les messages de façon asynchrone en vue d'un traitement ou pour les propager.

6. Procédé selon la revendication 5, dans lequel les communications entre les programmes d'application s'exécutant sur des systèmes informatiques différents du réseau comprennent au moins les étapes suivantes :

un premier programme d'application qui émet une instruction de placement de message sous la commande d'un gestionnaire de point de synchronisation dans les systèmes informatiques émetteurs, en vue d'émettre un message vers une file d'attente de messages,

des programmes de transmission de l'émetteur et du récepteur qui transfèrent des messages entre les systèmes informatiques, sous forme de deux unités de tâche liées logiquement, en utilisant des gestionnaires de points de synchronisation dans les deux systèmes informatiques d'émission et de réception, et

un second programme d'application qui émet une instruction de prise de message sous la commande d'un gestionnaire de point de synchronisation dans le système informatique de réception, afin de prélever le message de la file d'attente,

dans lequel les unités de tâche des instructions de placement de message, de transfert et de prise de message sont chacune maintenues dans l'incertitude jusqu'à la résolution de l'unité de tâche respective.

7. Procédé selon la revendication 1, dans lequel un message devant être délivré est émis vers une file d'attente à partir d'un programme d'application émetteur au niveau d'un premier ordinateur et est ensuite prélevé de façon asynchrone de la file d'attente devant être traitée par un programme d'application de réception, caractérisé en ce que :

chaque étape consistant à envoyer un message vers une file d'attente ou à prélever un message d'une file d'attente est exécutée sous la commande d'un programme de gestionnaire de file d'attente de messages, localisé au niveau d'un ordinateur du réseau,

les m ssages devant être délivrés à un programme d'application local sont placés sur une file d'attent locale desservie par le programme d'application local, alors que

des messages devant être délivrés à des programmes d'application distants sur des ordinateurs à distance sont placés sur des files d'attente de transmission locales en vue d'une transmission, respectivement pour chaque file d'attente de transmission, vers le programme de gestionnaire de file d'attente de messages suivant sur l'itinéraire vers les programmes de gestionnaire de file d'attente de messages distants de destination respectifs, dans lequel tous les messages placés sur une file d'attente de transmission particulière, lesquels messages peuvent être destinés à des programmes de gestionnaires de file d'attente de messages de destination différents, peuvent être transmis audit gestionnaire de file d'attente de messages suivant sous forme d'un ou plusieurs lots de messages à l'intérieur d'unités de tâche commandées par un gestionnaire de point de synchronisation.

8. Système de traitement de données comprenant un gestionnaire de messagerie destiné à des communications entre les programmes sur un réseau de systèmes de traitement de données, le gestionnaire de messagerie comprenant des programmes émetteurs et récepteurs destinés à transférer des messages entre des gestionnaires de messageries adjacents du réseau conformément au protocole de transfert suivant :

un programme émetteur d'un premier gestionnaire de messageries émettant un ou plusieurs messages à l'intérieur d'une première unité de tâche commandée par un gestionnaire de point de synchronisation,

un programme récepteur dans un second gestionnaire de messageries recevant lesdits messages à l'intérieur d'une seconde unité de tâche commandée par un gestionnaire de point de synchronisation,

les opérations d'émission et de réception étant maintenues dans l'incertitude, c'est-à-dire non engagées, jusqu'à la résolution des première et seconde unités de tâche, respectivement, et

caractérisé en ce que les première et seconde unités de tâche sont liées logiquement de façon à ce que l'engagement du traitement à la résolution des première et seconde unités de tâche comprend soit

(i) en réponse à la réception correcte des messages par l programme récepteur, l'engagement de ladite seconde unité de tâche, en transm ttant au programme émett ur une confirmation de réception positive, t en engageant en réponse à la confirmation positive la première unité de tâche, ou

(ii) en réponse à la réception incorrecte des messages, le renvoi en retour de la seconde unité de tâche, en transmettant au programme émetteur une confirmation de réception négative, et en restituant en réponse à ladite confirmation négative la première unité de tâche.

9. Système de traitement de données selon la revendication 8, dans lequel le gestionnaire de messageries est conçu pour des communications entre des programmes de mise en file d'attente de messages sur un réseau hétérogène de systèmes de traitement de données, le gestionnaire de messageries comprenant une interface de programmation d'application grâce à laquelle des applications se rattachent au gestionnaire de messageries et fournissant des services de mise en file d'attente permettant à des programmes d'application de placer des messages sur les files d'attente de messages en vue d'une récupération asynchrone par d'autres programmes d'application.

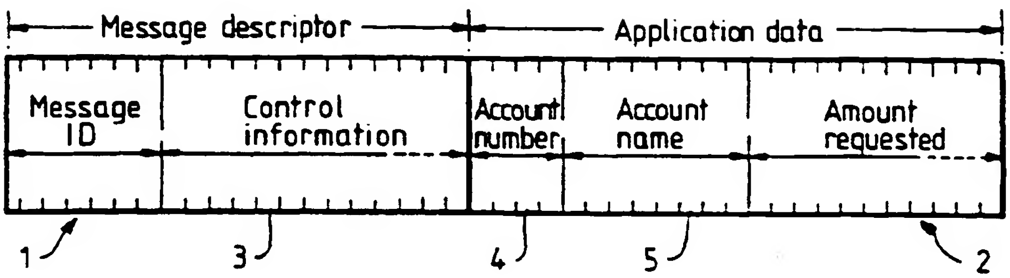
Message descriptor
Application data
Message ID
Control information
Account number
Account name
Amount requested
1
3
4
5
2

FIG. 1

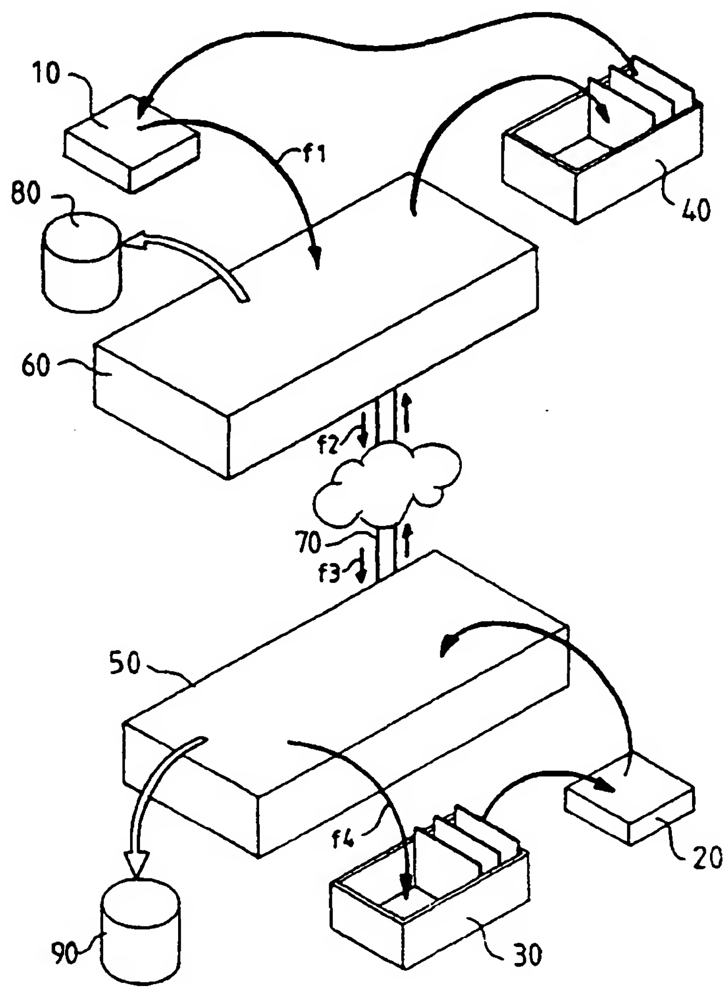
10
f1
80
40
60
f2
70
f3
50
f4
20
90
30

FIG. 2

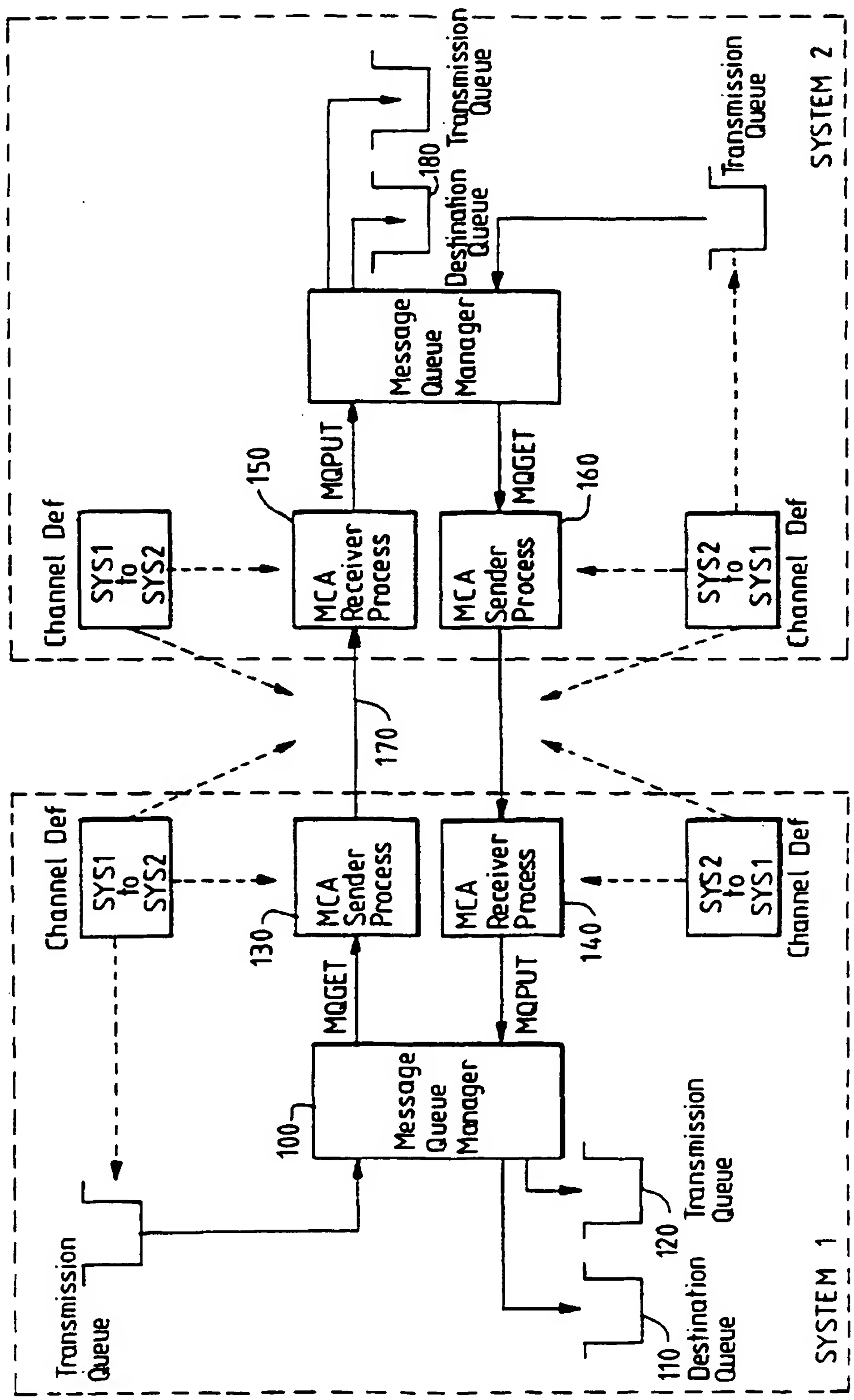
SYSTEM 1
SYSTEM 2
Transmission Queue
Message Queue Manager
100
MQGET
MQPUT
130
MCA Sender Process
140
MCA Receiver Process
Channel Def
SYS1 to SYS2
SYS2 to SYS1
Channel Def
110
Destination Queue
120
Transmission Queue
170
150
MCA Receiver Process
160
MCA Sender Process
MQPUT
MQGET
Message Queue Manager
180
Destination Queue
Transmission Queue
Transmission Queue
Channel Def
SYS1 to SYS2
SYS2 to SYS1
Channel Def

FIG. 3

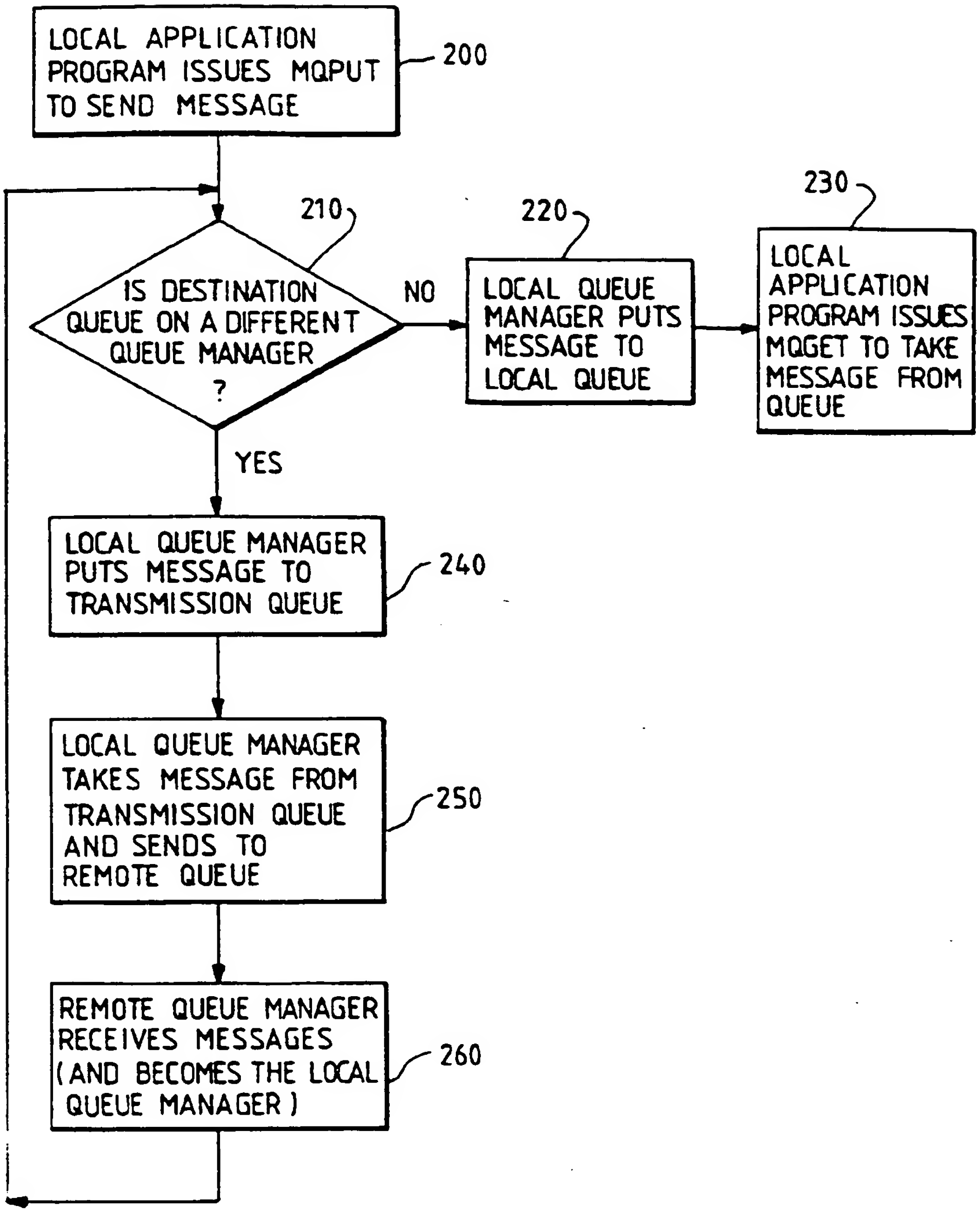
LOCAL APPLICATION PROGRAM ISSUES MQPUT TO SEND MESSAGE
200
210
IS DESTINATION QUEUE ON A DIFFERENT QUEUE MANAGER ?
NO
220
LOCAL QUEUE MANAGER PUTS MESSAGE TO LOCAL QUEUE
230
LOCAL APPLICATION PROGRAM ISSUES MQGET TO TAKE MESSAGE FROM QUEUE
YES
LOCAL QUEUE MANAGER PUTS MESSAGE TO TRANSMISSION QUEUE
240
LOCAL QUEUE MANAGER TAKES MESSAGE FROM TRANSMISSION QUEUE AND SENDS TO REMOTE QUEUE
250
REMOTE QUEUE MANAGER RECEIVES MESSAGES (AND BECOMES THE LOCAL QUEUE MANAGER)
260

FIG. 4

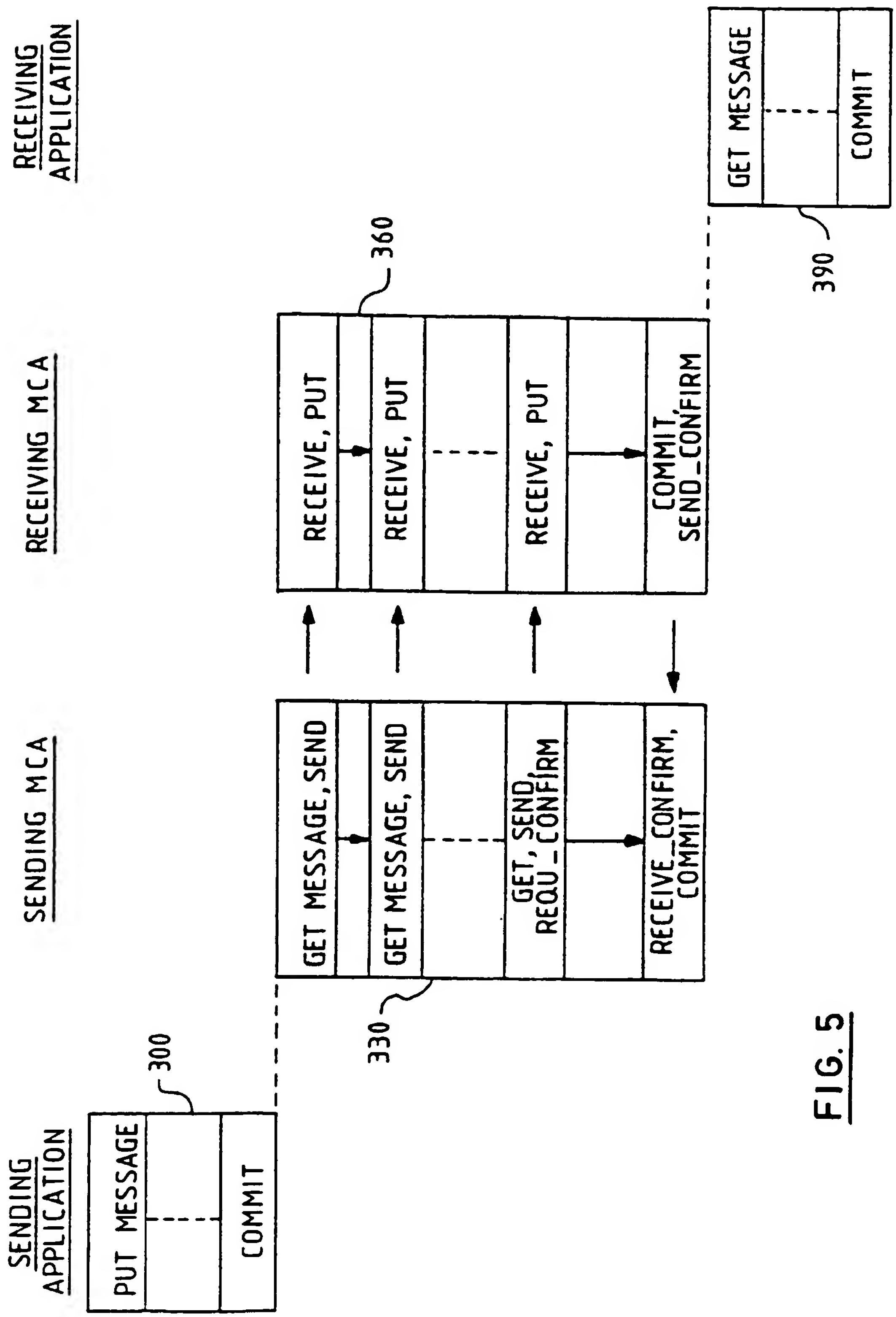
SENDING APPLICATION
SENDING MCA
RECEIVING MCA
RECEIVING APPLICATION
PUT MESSAGE
COMMIT
300
GET MESSAGE,SEND
GET MESSAGE, SEND
GET, SEND, REQU_CONFIRM
RECEIVE_CONFIRM, COMMIT
330
RECEIVE, PUT
RECEIVE, PUT
RECEIVE, PUT
COMMIT, SEND_CONFIRM
360
GET MESSAGE
COMMIT
390

FIG. 5